AR-004-140

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

ELECTRONICS RESEARCH LABORATORY

TECHNICAL REPORT

ERL-0330-TR

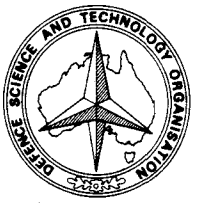
DIESS - A PROLOG IMPLEMENTATION OF A DOMAIN-INDEPENDENT EXPERT SYSTEM SHELL

R.F. Dancer

S U M M A R Y

A software environment suitable for the expression of expert knowledge in a variety of domains has been designed and implemented in the Prolog computer language. This 'shell' is intended to be capable of containing expert systems whose object is the proof of goal symbols in domains of categorical single-valued facts.

An example is given of a non-trivial expert system to programme in the IBM Job Control Language. Some conclusions are reached and recommendations given for future developments.



POSTAL ADDRESS: Director, Electronics Research Laboratory,
Box 2151, GPO, Adelaide, South Australia, 5001.

TABLE OF CONTENTS

Page

LIST OF APPENDICES

1. INTRODUCTION

This report describes the first version of a Domain-Independent Expert System Shell (DIESS Version 1) and initial experiences with its application to a 'real world' problem.

DIESS was built as an experimental tool needed (in lieu of other suitable software) as an essential part of a programme of research into the application of Artificial Intelligence (AI) to defence systems, and, in particular the use of 'Knowledge-Based' and 'Expert' systems for signal processing in Electronic Warfare systems. However, before entirely novel applications of the expert system technology to the arena of military operations (typically involving time-critical problems having outcomes with intrinsically high values) could be attempted it was thought wise to gain experience with applications having lower attendant technical and other risks. Nevertheless, the problem to be attempted should be of sufficient size and complexity to fully explore the expressive powers of the shell. The experimental expert system to programme in IBM Job Control Language (JCL) described below represents an attempt to apply the established expert system technology to one such area.

Expert systems are computer-based mechanisms concerned with the representation and application of the knowledge of human experts in specialised fields. Systems such as MYCIN (advising on diagnosis and therapy in cases of bacterial infection(ref.1)) and PROSPECTOR (consulting in Geology(ref.2)) have reached significant levels of performance - rivalling those of individual human specialists.

Experiences gained in these, and other, projects indicated the usefulness of strictly partitioning systems into two compartments; one containing knowledge specific to the domain of current interest and the other comprising essentially constant general-purpose reasoning machinery. This machinery, being independent of any field of specialisation, can then (it is theorised) be used to implement systems expert in any area by interpreting an appropriate knowledge base. The attractiveness of this concept was instrumental in the development of Essential MYCIN - known as EMYCIN(ref.3) and AL/X(ref.4) which are expert system 'shells' (ie systems emptied of their knowledge) ancestrally related to MYCIN and PROSPECTOR respectively. Additional confirmation of the utility of factoring systems in this way is supplied by (amongst others) the R1 system, used for planning configurations for computer hardware(ref.5). R1 was constructed from its inception as a specialised knowledge base utilising a pre-existing shell called OPS5(ref.6).

Due to various difficulties, in part related to the need to conserve resources, it has not, to date, been possible for the author to obtain and utilise any of the existing expert system shells to which reference has been made above. One pre-existing shell - the ASWE Expert System Support Programme(ref.7) - using the PROSPECTOR approach has been obtained and will be the subject of a later report.

The DIESS interpreter is conceptually related to EMYCIN in the structure of its facts and reasoning. The DIESS is intended to be a software shell forming a highly productive containing environment for rapid construction of backward-chaining expert systems which use categorical reasoning about single-valued facts.

Insofar as DIESS is concerned facts about an 'entity' (the subject of an examination) are established either by asking questions of the user (a human operator) or by deducing them from rules. The structure and content of the expert system knowledge base is rigorously defined by a domain expert, in conjunction with a knowledge engineer. This fund of knowledge is interpreted by DIESS to reason about the domain and produce facts in the data base relevant to a defined goal.

The work described herein was carried out under Task DST 82/251 - "Knowledge-Based ESM Processing"; and Task DST 83/213 - "Expert Systems Research".

2. IMPLEMENTATION

DIESS Version 1 is implemented in Portable Prolog(ref.8,9) - a language with powerful in-built pattern-matching and inference features. The DIESS interpreter and the expert system knowledge base are treated as sets of Prolog clauses. Protection against inadvertent corruption is afforded the entire expert system by auditing subsequent user commands to DIESS with an alternate interactive top level for Prolog called the 'Audited Command Shell Interpreter' (ACSI(ref.10)). This extra shell acts in a sentinel-like manner by allowing to pass through (for evaluation) only that user input which falls within a set of particular pre-defined forms. This auditing mechanism also guides the user as to the form of allowable input whilst, as is its primary function, securing the expert system shell and its knowledge and facts from accidental alteration.

3. THE KNOWLEDGE BASE

The domain-specific knowledge in an expert system is said to form a 'Knowledge Base'. So that the knowledge base together with the expert system shell form a complete expert system. The main component of the expert system shell is a 'reasoning machine' which 'thinks' about the knowledge base and user responses in order to solve the particular problem at hand. The human expert and the knowledge engineer thus have the task of expressing the domain expertise (or knowledge) in a language which can also be be understood by the reasoning machine. The syntax of the DIESS language (DIESSL) is formally defined using the Backus-Naur Form (BNF) in Appendix I. Following below is a brief description of the DIESS semantics attached to DIESSL knowledge forms.

3.1 Rules and facts

The fundamental units of information in DIESSL are (i) 'rules' and (ii) 'facts'. Facts may be true (ie established) either because the user knows them to be true (and communicates this knowledge to the expert system) or because their truth can be deduced from the combination of rules and previously established facts.

A rule expressed as a DIESSL statement has the general form:

RN : if ANTECEDENT

then

CONSEQUENT-FACT.

where **ANTECEDENT** is a single fact (or a logical function of a fact) or a conjunction of several facts (and/or of some logical functions of facts); **CONSEQUENT-FACT** is a hypothetical fact which is held to be true if the **ANTECEDENT** is true; and **RN** is a unique name for the rule. The rule name is used to identify the knowledge for administrative purposes: as a reference for editing, during explanation of chains of reasoning and in the detection and disqualification of circular reasoning. For the latter two uses, when a rule is invoked its name is pushed onto a stack of active rule names.

Facts in DIESSL all have the general form:

same(VARIABLE, VALUE)

where **VARIABLE** is a unique name for a means of classifying knowledge in the domain (it is often suitable to think of this as an attribute of a case in the domain); and **VALUE** is one of a set of (mutually exclusive) values which may be assigned to (ie made the **same** as) the **VARIABLE**. This form is used to convey the meaning that **VARIABLE** is the **same** as **VALUE**. The **VARIABLE** may be either non-numeric (ie have values which are members of a set of arbitrary objects called terms) or numeric (ie have values which are integers within a specified range).

3.2 Defaults

DIESSL also provides a means to define a default fact as:

DN : DEFAULT-FACT.

where **DEFAULT-FACT** is a hypothetical fact which is held to be true if no other value can be ascribed to its variable. The item **DN** is a unique name to identify the knowledge.

3.3 Logical functions allowed in an antecedent

The logical functions of facts, to which reference was made above, allowed to be incorporated in DIESSL antecedents are:

* **not same(VARIABLE, VALUE)**
* **unknown(VARIABLE)**
* **not unknown(VARIABLE)**
* **ask(VARIABLE, VALUE)**
* **greater(N-VARIABLE, N-VALUE)**
* **not greater(N-VARIABLE, N-VALUE)**
* **lesser(N-VARIABLE, N-VALUE)**
* **not lesser(N-VARIABLE, N-VALUE)**
* **between(N-VARIABLE, LOWER-N-VALUE, UPPER-N-VALUE)**
* **not between(N-VARIABLE, LOWER-N-VALUE, UPPER-N-VALUE)**

where **VARIABLE** and **VALUE** have the meanings given above; and **N-VARIABLE**, **N-VALUE**, **LOWER-N-VALUE** and **UPPER-N-VALUE** are connected exclusively with numeric facts.

3.4 Declaration of allowable facts

Each variable involved in a domain (ie in facts within the domain) must be declared in a DIESSL statement together with the criterion defining its value membership and its source. Such declarations have the form:

variable(**VARIABLE**, **MEMBERSHIP**, **SOURCE**).

where (i) **VARIABLE** is the (unique) name of the variable being declared; (ii) **MEMBERSHIP** is either: "members({val-1, val-2, ... })" - for non-numeric variables, or "limits(low, high)" - for numeric variables and (iii) **SOURCE** defines the ordinary source of the value for the variable. Variables involved in facts concluded by rules have the **SOURCE** "deduce", whilst others have the **SOURCE** "ask(TEXT)" - where TEXT is a literal character string (or a list of literal character strings) printed as a prompt to the user when a value is needed.

Variables having values to be obtained by deduction may be supplied with a value by the user, but only if the logical function "ask" appears in an antecedent of a rule executed before an attempt is made to conclude a value for the variable.

4. THE REASONING MACHINE

The reasoning machine interpreter - sometimes called an inference engine - is a vital component of the DIESS. It performs all logical processing by operating on the rules and auxiliary variable declarations in the knowledge base. A complete description of its functioning: how reasoning is initiated, how it is 'chained' to evaluate complex rules and how defaults are used follows below.

4.1 Initiation of reasoning

The operation of the reasoning machine is initiated by the user instructing it to evaluate a goal hypothesis - a putative fact comprising a goal variable and value (normally an unknown value). The reasoning machine has the task of attempting to ascribe a value to the variable. In so doing it firstly checks to see if a value is already known for the variable by looking in its data base for an established fact of the form it was instructed to prove. If there is already a value for the goal variable this occurrence is announced and the process finishes. If not, the reasoning machine attempts to deduce a value. In deducing a value the knowledge base is searched for rules which have any value of the goal variable in their consequent parts. Each such rule is evaluated in turn by attempting to prove its antecedent part. If the antecedent of a rule cannot be proven the rule is said to have 'failed' and the next rule is evaluated. If the antecedent is proven the rule and the goal are said to have 'succeeded'. Further rules (if any) for the goal are ignored, the conclusion of the rule is announced, and is added to the data base as an established fact before the process finishes. If no rule succeeds no conclusion is reached, the goal is said to have 'failed' and the process finishes.

4.2 Evaluation of antecedents, chaining and the data source

In evaluating the antecedent part of a rule - a conjunction of clauses - the reasoning machine will encounter clauses which are facts or are functions of facts. These clauses are evaluated as read, from left to right. Rules with antecedents which are single facts succeed if the antecedent is, or can be, established as true. Other rules require all antecedent clauses (ie the conjunction of clauses) to be established before they can succeed.

Antecedents are evaluated in two stages; firstly a 'preview' stage and secondly a 'chaining' stage.

4.2.1 The preview stage

Antecedent clauses may be either; (i) known to be true, or (ii) known to be false or (iii) neither known to be true nor known to be false. It is clear that rules having antecedents with at least one clause belonging to the second of these categories must fail, whilst those with all clauses belonging to the first must succeed. As the first stage of processing a rule which satisfies its current goal the reasoning machine therefore 'previews' each clause in the rule to see if it can be entirely disproven on the basis of existing internal knowledge; and duly fails the rule if so disproven. This feature prevents unnecessary processing and gathering of irrelevant information from the user. This is because, without it, a rule which is certain to fail at its second or any subsequent antecedent clause may, in evaluating a preceding clause, set itself a sub goal or require user response - both redundant actions. During the preview stage the reasoning machine constructs an edited antecedent by excluding those clauses which belong to category (i) above, hence only leaving the uncertain clauses - those belonging to category (iii). If it should happen that all clauses are edited out of the antecedent then the antecedent and the rule are deemed to have succeeded.

4.2.2 The chaining stage

If a rule passes the preview stage each clause in the edited antecedent is individually evaluated by the reasoning machine (in a chaining stage) by establishing such sub goals (ie new goals) as are necessary and referring these back to itself for examination - thereby re-initiating the reasoning machine. This recursive process is termed 'backward chaining'. If a sub goal of a rule fails then the antecedent cannot be proven and the rule fails. The reasoning machine examines the rules in a special order when a particular literal value is involved in a sub goal. Rules (if any) which conclude the particular literal value are the first to be examined. Other rules which include in their **CONSEQUENT-FACT** an unknown value (ie a variable value for the 'Variable') are examined subsequently. When the reasoning machine encounters a clause with a variable having the source "ask(TEXT)", if the variable has no known value, the reasoning machine prompts the expert system user to supply a value or to respond with the word "unknown". The latter response causes the clause involving "ask" to fail.

4.3 The use of defaults

In attempting to prove a clause of an antecedent; if all rules fail (or if a question designed to ascribe a value to a variable is answered "unknown") then the reasoning machine will look for and use the value in any default fact it may find in the knowledge base involving the current goal (or sub goal) variable.

5. OPERATION

An expert system (comprising DIESS and a knowledge base) is operated by running Prolog and 'consulting' files which contain the Prolog source for ACSI and DIESS, and the DIESSL source for the knowledge base.

The ACSI is an integrated top level interpreter which runs in either 'PROLOG' or 'ACSI' mode. The current implementation of DIESS utilises the ACSI mode for its top level. In this mode any prompt from the top level to the user may only be answered with one of the valid commands to which the user has access. (The command "help" is built into ACSI to assist users to establish which commands they may use). Access to the 'PROLOG' mode is available from ACSI only at the highest level of accreditation.

All DIESS operations are designed to occur through the ACSI after the user has been identified to the system during a login process (to ensure that individuals have appropriate levels of access to functions). Subsequent transactions at the top level of ACSI begin with the prompt:

***** ACSI >**

to which the user responds:

<command>.

where **<command>** is one of the commands defined to ACSI which allow user access to DIESS facilities. Explanations of these commands, including the 'examine' command, by which the DIESS reasoning machine is initiated, follow below.

5.1 The examine command

To commence operation of the expert system the user enters one of the following forms of the examine command:

examine IDENT.

examine IDENT for VALUE.

examine IDENT attribute VARIABLE for VALUE.

where **IDENT** is a unique name used to identify the results of the examination for later reference; **VALUE** is one of the members of the set of values of the goal variable; and **VARIABLE** is an alternative goal variable.

The first form of the **examine** command would mostly be employed to obtain a full examination of a case. However, partial examinations deducing any legal value of any defined variable as temporary goal, can be conducted using the second and third forms above. Examinations proceed in the form of question and answer sessions in which the expert system poses questions and the user answers them (with deductions normally being announced as they are made). The user may exercise close control over these sessions by entering a special command of a type specific to DIESS, a so-called 'internal directive', given in place of an answer to any question. A description of all internal DIESS directives, with their abbreviations in parentheses, is given below:

(a) why (w) - gives reasons why information is needed.

(b) suspend (s) - suspends the current examination.

(c) tracing (t) - institutes tracing of rules in use.

(d) notracing (nt) - discontinues rule tracing.

(e) annunciate (an) - institutes announcement of deductions.

(f) noannunciate (nan) - discontinues deduction announcement.

(g) explain (x) - lists available internal directives.

5.2 Other commands

Control of the rule execution trace and announcement of deductions is allowed from the top level by special ACSI commands. Tracing (which is normally off by default) and annunciation (normally on) can be turned on or off by the following commands:

trace_on.

trace_off.

annunciate_on.

annunciate_off.

Other commands can be defined to ACSI which operate on the fixed knowledge and on facts established by **examine** in useful ways. Such commands could for instance: summarise facts, explain how a deduction was made, and/or allow information to be volunteered. However, time constraints precluded the addition of these functions.

6. EXPERT JCL PROGRAMMER

This section describes the expert system which writes the JCL for an IBM 370 Job to read the first file of an unlabelled magnetic tape and copy it to a file. The expert system encompasses the full range of tape forms physically acceptable at the DRC Computing Centre installation. It was chosen for implementation as an expert system because it has a domain of reasonable size and complexity and the expertise to build it was close at hand.

6.1 Rationale

The system was built primarily to explore the capacity of DIESS to express and process expertise at a non-trivial level in a narrow quasi-static domain. However the possibility of ultimately providing a useful working system was not overlooked.

Composing the JCL code to instruct the reading of a magnetic tape at a large time-sharing computer installation is an oft-repeated and time-consuming task. It requires that the programmer have a moderate amount of knowledge which may only be gleaned from a wide variety of approximately static discrete sources. Such a feat of memory is difficult and unnecessary for clients who have only a small requirement to read tapes and and whose main professional interest is other than in computer 'housekeeping'.

The composition of such special-purpose JCL would thus seem to be a prime candidate for automation as an expert system. It could then, if suitably efficient, be made available to usefully advise a large community of computer users who occasionally need to import non-indigenous information by tape to an IBM 370 host computer (eg special-purpose software applications or experimental data).

6.2 JCL programming knowledge base

The JCL knowledge base, expressed in DIESSL was compiled substantially by the author, acting as both knowledge engineer and as domain expert. In this latter capacity reference was made to published material of varying degrees of formality (detailed in references 11 to 14 below) and to memories of some sparse personal experience with similar problems.

The knowledge base initially took some 3 days to write, followed by 5 or so days of intensive debugging to perfect: changes made in response to criticisms mentioned below took another 2 days to incorporate.

A total of 50 rules, 14 default facts and 54 variable declarations are included in the knowledge base. Of the 54 variables used, 41 relate to deduced facts and 13 to asked facts.

The authority for discrete items of knowledge is given in comment preceding that knowledge. Reference is made therein to: definitive IBM publications(ref.11,12) concerning facilities of the IBM OS operating system; a fixed local installation standard - promulgated in reference 13; and to a somewhat more fluid installation standard given in reference 14.

The knowledge base was written with the aim of being self-explanatory and to this end included copious comment text. Taking approximately 850 lines of DIESSL code, it was considered too voluminous to reproduce in full in this document. Certain extracts of the knowledge base for the JCL programming expert system (given in Appendix II) have been selected for particular comment.

The extracts, comprising some 240 lines of code, have been selected to illustrate typical, as well as notable, uses of DIESSL, with examples of major features of the language. They include; a full explanation of the top level goal, general illustrative features of the domain knowledge and examples of special-purpose native Prolog code.

6.2.1 Top level objective

As a basic knowledge engineering objective it was decided that the expert system should at all material times give the user the maximum amount of pertinent information about each tape examination. This requirement implied production of:

(a) explanation of when the task was impossible,

(b) indication of when essential primitive data was not available and

(c) output of JCL in a form suitable for submission to JES2 (the Job Entry Subsystem of OS) for execution.

This analysis of objectives is, in fact, a prescription for the top level goal of the JCL expert system given in Appendix II.1.

The DIESSL text defining the top level goal comprises a declaration for variable 'result' and three rules 'c1', 'c2', and 'c3'. These rules each have an 'action' clause to execute the functions outlined above. The actions required to be taken by rule 'c1' include a reference to 'make_fn', a predicate containing some special-purpose Prolog code to construct a unique file name. Rule 'c2' contains an example of a fact with an incompletely 'instantiated' (ie partially unformed) variable 'missing(P)' (in which P would contain the name of any missing parameter designated as essential). The use of these structured variables is discussed below in more detail. Rule 'c3' gives an example of the use of the logical function 'unknown'.

6.2.2 A first level sub goal

One of the sub goals in rule 'c1' above is that the tape being examined is 'physically standard' - ie is physically compatible with the available tape unit hardware. The knowledge which defines this aspect is given in Appendix II.2. This knowledge comprises declarations for three variables and two rules in DIESSL.

Values for the variable 'physically_standard_tape', are deduced by each of the rules 'p1' and 'p2a'. The variables 'ordinary_looking_tape' and 'visually_checked_ok' relate to primitive data, values for which are acquired directly from the operator, after posing the appropriate question. The antecedent of rule 'p2a' contains another example of the use of the logical function, 'unknown'.

6.2.3 JCL programme structure

Another of the sub goals in rule 'c1' transmits the JCL programme produced by the expert system to its action clause as the variable term 'JCL'. The knowledge which defines this programme structure is given in Appendix II.3. This information divides the complete job naturally into its various statements in their correct order.

The variables 'jcl_prog' and 'prog_body' are deduced by the rules 'jcl0' and 'jcl1' respectively. Each of the rules builds up a collection of programme parts and uses the ad hoc predicate 'join' to concatenate them as the last subgoal of its antecedent.

6.2.4 JCL statement construction

The knowledge about how to construct a JCL statement (used in rules 'jcl0' or 'jcl1' as discussed above) is exemplified by that given in Appendix II.4 for the SYSUT1 DD statement of the IEBGENR IBM utility. A prototype form of the statement (with symbolic parameters preceded by dollar signs) together with a list associating the parameter names with their actual (deduced or asked) values is processed by an ad hoc Prolog predicate 'symb_subs', to give the completed form of the statement.

Two rules, 'da1' and 'da2' conclude the completed statement and build the association list (as the values of the variables, 'sysut1' and 'jcl_sysut1_parms') respectively. A third variable declaration serves to convey knowledge about the logical record length of the file on the input tape, which is the subject of one of the sub goals in rule 'da2'. This primitive data is passed via the structured variable 'p(_)' discussed below.

6.2.5 Data collection

There are many items of information which are vital to the successful completion of the JCL programme. If any one of these is missing it should be notified to the top level. This was accomplished by passing all vital parameter values via structured variables of the form 'p(Y)' for numeric parameters and 'q(Y)' for non-numeric parameters, where Y carries the parameter name. If no value can be assigned to a parameter so treated the fact that 'missing(Y)' is true is recorded. The three rules ('uua', 'uub' and 'uuc') and three variable declarations (for 'p(Y)', 'q(Y)' and 'missing(Y)') shown in Appendix II.5 give the knowledge to perform these functions.

6.2.6 Special-purpose code

A few additional ad hoc (ie non-DIESS) Prolog predicates were needed to cope with the JCL writing domain. Amongst these were predicates to accomplish symbolic parameter substitution in prototype JCL code ('symb_subs') and the efficient concatenation of more than two lists ('join'). The Prolog definition of 'symb_subs' is given in Appendix II.6 below. A test on the operation of this predicate showed that it consumed 1.09 s CPU time for the 'sysut1' example referred to above.

Criticism of the JCL writing expert system (in terms of the contents of its knowledge base and its operation) was invited from persons(ref.15) with more experience in this use of JCL than the author. Some suggestions were subsequently incorporated in the knowledge base, and others repeated (in general terms) in recommendations below.

Further refinements would be needed to develop the knowledge base to a suitable standard for production use. One refinement, to increase efficiency, could be to relegate the symbolic substitution of parameters in prototype JCL to a TSO command list.

7. EXAMPLE TAPE EXAMINATIONS

The JCL expert system was run to demonstrate its operation for the three possible goal conclusions:

(i) same(result, task_impossible),

(ii) same(result, task_incomplete), and

(iii) same(result, task_complete)

which correspond with the basic objectives given above.

The examination output for each of these conditions is given in Appendix III. The third examination successfully constructed a JCL programme, printing it and copying it to an external file.

As can be seen these examples took 0.82 s, 1.80 s and 35.13 s of host CPU time respectively on the IBM 370/3033, one of the larger and faster computers in common use. This level of resource consumption is currently unacceptable for other than highly experimental applications. The reasons for such a large CPU time appetite are several, the foremost being that a great deal of processing is unavoidably incurred on each occasion that a Prolog goal is established and there is a large database to search. The second significant cause is that Portable Prolog is itself implemented in Pascal. Despite optimised compilation with the IBM Pascal/VS compiler execution of the Portable Prolog object code programme is very slow.

Storage statistics displayed indicate that storage used by the reasoning machine to make temporary associations during the operation of DIESS (the "heap" space) grew irreversibly. This space was not recovered after control was returned to the top level of the Audited Command Shell Interpreter. In any practical application of this system occupied storage would thus continue to grow until the maximum available storage was exceeded. Experience with this knowledge base has shown that an attempt to make a second successful examination in the DIESS run listed in Appendix III would have caused Prolog to 'crash' due to requests for excessive amounts of main storage. (DIESS was operating with Prolog and TSO in a region of 2 Mbytes!) The problem of exhausting available storage is primarily caused by the lack of a 'garbage collector' routine within the Version 1 Portable Prolog implementation - a facility which would reclaim discarded heap space and make it available for re-use.

8. RECOMMENDATIONS

Before DIESS (or the DIESS/JCL expert system) could be qualified for general use action would be needed to:

(a) decrease consumption of CPU time,

(b) decrease main storage use and

(c) increase 'user friendliness'.

These areas could, in part, be fruitfully addressed by:

(a) Acquiring a faster version of Prolog for the DRC IBM 370 - such as Waterloo Prolog(ref.16);

(b) Investigating the possibility of re-coding DIESS in Standard LISP(ref.17) to take advantage of higher execution speeds obtained with compiled code;

(c) Re-designing DIESS to decrease non-essential run-time syntax checking and/or allow pre-compilation of DIESSL rules into clauses directly executable by Prolog;

(d) In-house modification (or acquisition of a later version) of Portable Prolog implementing garbage collection;

(e) Implement ACSI/DIESS commands to allow inquiries to be made about what existing knowledge there is about a particular case;

(f) Implement an ACSI/DIESS 'how' command to answer queries about how certain deductions were made;

(g) Implement an ACSI/DIESS command to allow the user to volunteer facts.

9. CONCLUSION

A domain-independent expert system shell has been written in the Prolog computer language. This shell preserves the generality of Prolog expressions for use in extreme cases whilst providing a standard framework for rapid construction of goal-directed expert systems using categorical reasoning.

It has been demonstrated that DIESS is well capable of expressing the knowledge of a non-trivial domain, with very little special-purpose Prolog code.

REFERENCES

No.	Author	Title
1	Davis, R. Buchanan, B.G. and Shortliffe, E.H.	"Production-rules as a Representation of a Knowledge-based Consultation Program". Artificial Intelligence, 8, 15-45, 1977
2	Duda, R.O.	"The PROSPECTOR System for Mineral Exploration". Final Report, SRI Project 8172, SRI International, 1980
3	van Melle, W.	"A Domain Independent System that Aids in Constructing Knowledge-based Consultation Programs". Stanford University STAN-CS-80-820, 1980
4	-	"AL/X Computer Software Description". Intelligent Terminals Ltd., 15 Canal St., Jericho, Oxford, 1983
5	McDermott, J.	"R1: A Rule-Based Configurer of Computer Systems". Artificial Intelligence 19, 39-88, 1982
6	Forgy, C.L.	"OPS5 User's Manual". Report CMU-CS-81-135, Department of Computer Science, Carnegie Mellon University, July 1981
7	Miles, J.A.H.	"Preliminary Report on Expert Systems and the User's Guide for the ASWE Expert System Support Programme". Admiralty Surface Weapons Establishment, Memorandum XCC 82006, March 1982
8	Clocksin, W.F. and Mellish, C.S.	"Programming in Prolog". Springer-Verlag, 1981
9	Spivey, J.M.	"University of York Portable Prolog System, Release 1, User's Guide". Software Technology Research Centre, University of York, March 1983
10	Dancer, R.F.	"An Audited Command Shell Interpreter for Prolog Programmes". Systems Studies Group Working Paper. ES-03, September 1984
11	-	"OS VS/2 JCL VS2 Release 3 - Second Edition". International Business Machines Corporation. GC28-0692-1, February 1975

No.	Author	Title
12	-	"OS VS/2 MVS Utilities Release 3.8 - First Edition". International Business Machines Corporation. GC26-3902-0, December 1977
13	Shaw, P.	"UNIT Names for Tape Units". Weapons Research Establishment, Computing Services Group. Computer Bulletin No. 87, May 1977
14	-	"TSO HELP CLASSES Interactive Command". DRCS IBM370 Installation Standard, July 1984
15	Willcocks, P.J.	Personal communication, August 1984
16	Roberts, G.	"Waterloo PROLOG User's Manual - Version 1.4". Department of Computer Science, University of Waterloo, 1983
17	Marti, J.B. Hearn, A.C., Griss, M.L. and Griss, C.	"Standard Lisp Report". University of Utah, UUCS-78-101, 1978
18	Gries, D.	"Compiler Construction for Digital Computers". Wiley, 1971

APPENDIX I

DIESSL SYNTAX

The formal definition of a language syntax (ie its grammar) can be expressed conveniently in the so-called Backus-Naur Form (BNF). Legal constructs in a language are defined with BNF by associating 'non-terminal' symbols with collections of other 'non-terminal' symbols and 'terminal' symbols by the 'meta-symbols' "::=" and "|" to form 'productions'. A hierarchical non-circular structure of these productions is usually defined so that a top or goal symbol (not appearing on the right hand side of any production) is ultimately defined only in terms of terminal symbols (by substituting for lower level non terminals in the right hand sides of higher level productions) A more complete dissertation on the BNF method may be found in reference 18.

Following below are definitions of the grammar and vocabulary of DIESSL and the form of comments.

I.1 BNF grammar productions for DIESSL

DIESSL is a free form language in which the parser only recognises a stream of symbols separated by spaces and other characters (including the 'end-of-line' token).

```
<Expert System> ::= <System Id> <Knowledge List>

<System Id> ::= <System Name> <Goal>

<System Name> ::= signature( <Signature> , <Domain Name> ).

<Signature> ::= <Text>

<Domain Name> ::= <Constant>

<Goal> ::= goal( <Variable Name> ).

<Knowledge List> ::= <Knowledge Element> <Knowledge List>
                   | <Knowledge Element>

<Knowledge Element> ::= <Variable Declaration>
                      | <Rule Declaration>
                      | <Default Declaration>

<Variable Declaration> ::= variable( <Variable Name> ,
                                     <Values> ,
                                     <Information Source> ).

<Rule Declaration> ::= <Rule Name> : if <Antecedent Term>
                                     then
                                        <Consequent Term> .

<Default Declaration> ::= <Rule Name> : <Consequent Term> .

<Variable Name> ::= <atom>
                  | <structure>

<Values> ::= members({ <Member List> })
           | limits( <Lower Limit> , <Upper Limit> )

<Member List> ::= <Member> , <Member List>
```

```
                | <Member>

<Member> ::= <atom>
           | <structure>
           | <list>
           | <variable>

<Lower Limit> ::= <Number>

<Upper Limit> ::= <Number>

<Number> ::= <integer>
           | <variable>

<Information Source> ::= ask( <Text Specification> )
                       | deduce

<Text Specification> ::= <Text>
                       | { <Text List> }

<Text List> ::= <Text> , <Text List>
              | <Text>

<Text> ::= <atom>

<Rule Name> ::= <atom>

<Antecedent Term> ::= { <Antecedent Clause List> }

<Antecedent Clause List> ::= <Antecedent Clause> ,
                                    <Antecedent Clause List>
                           | <Antecedent Clause>

<Antecedent Clause> ::= <'same' Clause>
                      | <'unknown' Clause>
                      | <'greater' Clause>
                      | <'lesser' Clause>
                      | <'between' Clause>
                      | <'ask' Clause>
                      | <clause>

<'same' Clause> ::= <Same Function>
                  | not <Same Function>

<Same Function> ::= same( <Variable Name> , <Value Symbol> )

<'unknown' Clause> ::= <Unknown Function>
                     | not <Unknown Function>

<Unknown Function> ::= unknown( <Variable Name> )

<'greater' Clause> ::= <Greater Function>
                     | not <Greater Function>

<Greater Function> ::= greater( <Variable Name> , <Number> )

<'lesser' Clause> ::= <Lesser Function>
                    | not <Lesser Function>

<Lesser Function> ::= lesser( <Variable Name> , <Number> )
```

```
<'between' Clause> ::= <Between Function>
                     | not <Between Function>

<Between Function> ::= between( <Variable Name> ,
                                 <Number> , <Number> )

<'ask' Clause> ::= ask( <Variable Name> , <Value Symbol> )

<Value Symbol> ::= <Member>
                 | <Number>

<Consequent Term> ::= <Same Function>
                    | <Same Function> <Consequent Action>

<Consequent Action> ::= action { <Consequent Action list> }

<Consequent Action list> ::= <clause> , <Consequent Action list>
                           | <clause>

<Constant> ::= <atom>
             | <integer>

<Term> ::= <Constant>
         | <structure>
         | <list>
         | <variable>
```

I.2 Vocabulary

The vocabulary of a grammar comprises the sets of non-terminal symbols and terminal symbols. These two sets for DIESSL are enumerated and discussed below.

I.2.1 Non-terminal symbols

Non-terminal symbols of the DIESSL vocabulary are those which appear in productions on the left hand side of the 'replacement' (ie "::=") BNF meta symbol. Their syntactic content is defined by the BNF grammar. The non-terminal symbols in DIESSL are listed below:

<Expert System>	<Rule Name>
<System Id>	<Antecedent Term>
<System Name>	<Antecedent Clause List>
<Signature>	<Antecedent Clause>
<Domain Name>	<'same' Clause>
<Goal>	<Same Function>
<Knowledge List>	<'unknown' Clause>
<Knowledge Element>	<Unknown Function>
<Variable Declaration>	<'greater' Clause>
<Rule Declaration>	<Greater Function>

<Default Declaration>	<'lesser' Clause>
<Variable Name>	<Lesser Function>
<Values>	<'between' Clause>
<Member List>	<Between Function>
<Member>	<'ask' Clause>
<Lower Limit>	<Value Symbol>
<Upper Limit>	<Consequent Term>
<Number>	<Consequent Action>
<Information Source>	<Consequent Action list>
<Text Specification>	<Constant>
<Text List>	<Term>
<Text>	

I.2.2 Terminal symbols

Terminal symbols in the DIESSL syntax are those symbols which do not appear on the left hand side of productions. They are (i) literal characters and (ii) composite symbols appearing within angular brackets "<" and ">". Brief descriptions of the six terminal symbols in this latter category (viz <atom>, <integer>, <list>, <variable>, <structure> and <clause>) appear immediately below. These objects are native to Prolog and are more fully described in references 8 and 9.

I.2.2.1 <atom>

The symbol <atom> stands for one, or a sequence of, non-blank alphamerics (including underscores) commencing with a lower case alphabetic; or any sequence of characters enclosed in single quotes (with a single quote character embedded in the string being represented by two quotes in sequence).

I.2.2.2 <integer>

The symbol <integer> is composed of one, or a sequence of, numeric characters, optionally preceded by a sign (+ for a positive number - the default, and ~ for a negative number).

I.2.2.3 <list>

The symbol <list> stands for a Term, or a sequence of Terms separated by commas (or head and tail Terms separated by |) enclosed in braces (ie {}).

I.2.2.4 <variable>

The symbol <variable> stands for one, or a sequence of, non-blank alphamerics commencing with an upper case alphabetic or underscore character.

I.2.2.5 <structure>

The symbol <structure> stands for an atom (called the principal functor) followed by a sequence of of arguments contained in parentheses and separated by commas, where each argument is in its most general sense a Term (ie <Term>). Although <structure> is thus not a truly terminal symbol, no substantive difficulties are incurred by this minor inconsistency.

I.2.2.6 <clause>

The symbol <clause> stands for an expression to be established as a Prolog goal.

I.3 Comments

Comments in DIESSL may appear anywhere between terminal symbols. Comment text commences with the "/*" and ends with the "*/" composite symbols.

APPENDIX II

EXTRACTS OF JCL PROGRAMMER EXPERT SYSTEM KNOWLEDGE BASE

II.1 Top level knowledge

```
/* ------------------------------------------------------ */
/*  Knowledge to establish top-level goals                */
/*                                                        */
/* Author : RFD                                           */
/* Ref : July'84, amended Sept '84.                       */
/* ------------------------------------------------------ */

variable(result,
         members({task_complete,
                  task_incomplete,
                  task_impossible}),
         deduce).

c1: if { same(physically_standard_tape, yes),
         same(jcl_prog, JCL),
         same(q(volser), VOLSER) }
    then
       same(result, task_complete)
       action {nl,
          write(' I have written the following JCL programme '),
          writeln('to read your tape:'),
          nl,
          writelist(JCL),
          make_fn(VOLSER, FN),
          tell(FN),
          writelist(JCL),
          told,
          nl,
          write(' This code has been copied to file '),
          write(FN),
          writeln('.'),
          nl}.

c2: if { same(physically_standard_tape, yes),
         same(missing(P), true) }
    then
       same(result, task_incomplete)
       action {nl,
 write('  You will need to discover'),
 writeln(' and supply me with the value of'),
 write(' '), writeln(P),
 writeln(' before I can write the JCL to read your tape.'),
 writeln('  I will generally need to know the tape recording density,'),
 writeln(' number of tracks, recording mode, record format, block '),
 writeln(' size and record length in order to write the JCL.'),
 writeln('  The TSO command ''tapemap'' may be useful for getting'),
 writeln(' some of these values.'),
       nl}.
```

```
c3: if { unknown(physically_standard_tape) }
    then
       same(result, task_impossible)
       action {nl,
 writeln('  It seems that your tape is not physically compatible'),
 writeln(' with the IBM tape drives. You should check with'),
 writeln(' CS group to positively confirm this conclusion.'),
 writeln(' Sorry I can''t help further.'),
        nl}.
```

II.2 Some first level knowledge

```
/* ---------------------------------------------------- */
/*  Knowledge to establish tape is physically amenable  */
/*                                                      */
/* Author : RFD - July, 1984.                           */
/* Ref : July'84                                        */
/* ---------------------------------------------------- */


variable(physically_standard_tape,
         members({yes}),
         deduce).

p1: if {same(ordinary_looking_tape, yes)}
    then
       same(physically_standard_tape, yes).

p2a: if { unknown(ordinary_looking_tape),
          same(visually_checked_ok, yes) }
    then
       same(physically_standard_tape, yes).

variable(ordinary_looking_tape,
         members({yes, no}),
         ask('Is it an ordinary-looking tape')).
variable(visually_checked_ok,
         members({yes, no}),
         ask({
'Is it a single reel of tape 0.5 in. wide with an external diameter of',
'between 7.0 and 10.5 in. and an internal diameter of about 3.75 in.'
            })).
```

II.3 JCL programme structure

```
/* ---------------------------------------------------- */
/*  Knowledge to build JCL programme                    */
/*                                                      */
/* Author : RFD - July, 1984.                           */
/* Ref : OS/VS2 JCL (Rel 3), p19                        */
/* ---------------------------------------------------- */

variable(jcl_prog, members({{{_}|_}}), deduce).

jcl0: if { same(prog_body, B),
           same(job_stmt, J),
           same(comment, C),
           join({J, C, B}, JCB) }
      then
         same(jcl_prog, JCB).
```

```
/* ------------------------------------------------------ */
/*  Knowledge to build body of programme                  */
/*                                                        */
/* Author : RFD - July, 1984.                             */
/* Ref : OS/VS2 MVS Utilities, p 9-5, p9-15               */
/* ------------------------------------------------------ */

variable(prog_body, members({{{_}|_}}), deduce).

jcl1: if { same(sysut1, D1),
           same(sysut2, D2),
           same(exec_stmt, E),
           same(sysprint, D3),
           same(sysin, D4),
           same(terminator, TE),
           join({E, D1, D2, D3, D4, TE}, B) }
      then
         same(prog_body, B).
```

II.4 JCL DD statement construction

```
/* ------------------------------------------------------ */
/*  Knowledge to establish DD stmts for IEBGENER exec     */
/*                                                        */
/* Author : RFD - July, 1984.                             */
/* Ref : OS/VS2 MVS Utilities, p 9-15 &                   */
/*       OS/VS2 (Rel 3) JCL.                              */
/* ------------------------------------------------------ */

prototype_sysut1({
"//SYSUT1 DD UNIT=$unit,VOL=SER=$volser,DISP=OLD,",
"//       DCB=(RECFM=$recfm,LRECL=$lrecl$blksize_p,",
"//            DEN=$den$optcd_p$trtch_p),",
"//       LABEL=(,NL)"
          }).

variable(sysut1, members({{{_}|_}}), deduce).

da1: if { same(jcl_sysut1_parms, Parms),
          (prototype_sysut1(Syp), symb_subs(Parms, Syp, Sy)) }
     then
        same(sysut1, Sy).
```

```
variable(jcl_sysut1_parms, members({{H|T}}), deduce).

da2: if { same(unit, UNIT),
          same(q(volser), VOLSER),
          same(p(logical_record_length), LRECL),
          same(recfm, RECFM),
          same(blksize_p, BLKSIZE_P),
          same(den, DEN),
          same(optcd_p, OPTCD_P),
          same(trtch_p, TRTCH_P) }
     then
        same(jcl_sysut1_parms, { unit(UNIT),
                                 volser(VOLSER),
                                 recfm(RECFM),
                                 lrecl(LRECL),
                                 blksize_p(BLKSIZE_P),
                                 den(DEN),
                                 optcd_p(OPTCD_P),
                                 trtch_p(TRTCH_P)
                               }).

variable(logical_record_length, limits(1, 32768),
         ask('What is the record length (in bytes)')).
```

II.5 The collection of vital parameters

```
/* ------------------------------------------------------ */
/*  Rules to supply parameters and record unknowns        */
/*                                                        */
/* Author : RFD - July, 1984.                             */
/* ------------------------------------------------------ */

variable(p(_), limits(_, _), deduce).

uua: if { unknown(missing(Y)),
          same(Y, Z) }
     then
        same(p(Y), Z).

variable(q(_), members({ _ }), deduce).

uub: if { unknown(missing(Y)),
          same(Y, Z) }
     then
        same(q(Y), Z).

variable(missing(_), members({ true }), deduce).

uuc: if { unknown(Y) }
     then
        same(missing(Y), true)
        action {nl,
          write(' The value of essential parameter: '),
          write(Y),
          writeln(' is missing.'),
          nl}.
```

II.6 Prolog code for symbolic substitution

```
/* substitution for parameters in Spec */

symb_subs(Pms, Lines, Procd_Lines) :-
   Stc is "$",
   symb_subst({Stc|Pms}, Lines, Procd_Lines).

symb_subst(Specs, {Line_1|Lines}, {Procd_Line_1|Procd_Lines}) :-
   symb(Specs, Line_1, Procd_Line_1),
   symb_subst(Specs, Lines, Procd_Lines), !.
symb_subst(_, {}, {}).

symb(Specs, Line, Procd_Line) :-
   found_start_char(Specs, Line, Procd_Line), !.
symb(Specs, {C1|Cs}, {C1|PCs}) :-
   symb(Specs, Cs, PCs), !.
symb(_, {}, {}).

found_start_char({Stc|Pms}, {Stc|Line}, Procd_Line) :-
   word_in_string_head(Line, Residue, W_input_string),
   replacement(W_input_string, Pms, Word_string),
   symb({Stc|Pms}, Residue, Procd_Residue),
   append(Word_string, Procd_Residue, Procd_Line), !.

word_in_string_head({C1|Cs}, Res, {C1|Ws}) :-
   not delim(C1), !,
   word_in_string_head(Cs, Res, Ws).
word_in_string_head({}, {}, {}) :- !.
word_in_string_head(Cs, Cs, {}) :- !.

delim(C) :-
  dlm(Cs),
  C is Cs.

dlm(" ").
dlm(",").
dlm(".").
dlm(")").
dlm("$").

replacement(Wins, Pms, Wouts) :-
   name(Wm, Wins),
   rv(Wm, Pms, Wouts).

rv(Wm, {WIO|_}, Wouts) :-
   WIO =.. {Wm, Watomic}, !,
   xname(Watomic, Wouts).
rv(Wm, {_|WIOs}, Wouts) :-
   rv(Wm, WIOs, Wouts).
rv(Wm, {}, "") :-
   write(' Symbolic parameter :'),
   write(Wm),
   writeln(' is unknown.').
```

APPENDIX III

THREE EXAMPLE TAPE EXAMINATIONS

```
 READY
prolog preload(rootjcl3) sees(test)
 Portable Prolog Release 1.

  DIESS Version 1.0 R2/JCL programmer (tape reading) V1.1
  DATE: 09/18/84     TIME: 13:50:06.
  enter login(userid, pwd) or end.
login(rfd,xxx).
  *** User rfd is logged in to ACSI for access at level: admin.
 yes
  >
call(stats).

 Prolog Execution Statistics:
 Local Frames: 14, Variables: 10
 Global Nodes: 11
 Heap Nodes: 12241
 Trail Entries: 2
 Atoms: 863, Chars: 9411

 yes
```

III.1 Examination of a 'Bad' Tape

```
  >
 /*   Example No. 1.      Bad Tape   */


examine t0.


  *** Initiating examination for "t0".
  EXPERT SYSTEM: JCL programmer (tape reading) V1.1
  DATE: 09/18/84     TIME: 13:51:55.
  ----------------------------------

  Is it an ordinary-looking tape
  (Give one of: yes, no,  or unknown).?
unknown.

  Is it a single reel of tape 0.5 in. wide with an external diameter of
 between 7.0 and 10.5 in. and an internal diameter of about 3.75 in.
  (Give one of: yes, no,  or unknown).?
no.
```

```
 It seems that your tape is not physically compatible
with the IBM tape drives. You should check with
CS group to positively confirm this conclusion.
Sorry I can't help further.

*** Concluded "t0" result task_impossible using rule c3.
*** Terminating examination for "t0".
yes
 >
dt.
*** Execution time was 817 mS.
yes
 >
call(stats).

Prolog Execution Statistics:
Local Frames: 14, Variables: 10
Global Nodes: 11
Heap Nodes: 12425
Trail Entries: 2
Atoms: 865, Chars: 9421
yes
```

III.2 Examination with insufficient information

```
 >
/*   Example No. 2.      Insufficient information  */


examine t1.


*** Initiating examination for "t1".
EXPERT SYSTEM: JCL programmer (tape reading) V1.1
DATE: 09/18/84     TIME: 13:53:09.
---------------------------------

Is it an ordinary-looking tape
(Give one of: yes, no,  or unknown).?
yes.
*** Deduced "t1" physically_standard_tape yes using rule p1.

Is it a 7- or 9-track tape
(Give one of: 7, 9,  or unknown).?
unknown.

The value of essential parameter: number_of_tracks is missing.

*** Deduced "t1" missing(number_of_tracks) true using rule uuc.

 You will need to discover and supply me with the value of
number_of_tracks
before I can write the JCL to read your tape.
 I will generally need to know the tape recording density,
number of tracks, recording mode, record format, block
size and record length in order to write the JCL.
 The TSO command 'tapemap' may be useful for getting
some of these values.
```

```
  *** Concluded "t1" result task_incomplete using rule c2.
  *** Terminating examination for "t1".
 yes
  >
dt.
  *** Execution time was 1798 mS.
 yes
  >
call(stats).

 Prolog Execution Statistics:
 Local Frames: 14, Variables: 10
 Global Nodes: 11
 Heap Nodes: 12906
 Trail Entries: 2
 Atoms: 867, Chars: 9431
 yes
```

III.3 Successful examination

```
  >
 /*   Example No. 3.      Completes JCL   */


examine t2.


  *** Initiating examination for "t2".
  EXPERT SYSTEM: JCL programmer (tape reading) V1.1
  DATE: 09/18/84     TIME: 13:54:48.
  ----------------------------------

  Is it an ordinary-looking tape
  (Give one of: yes, no,  or unknown).?
yes.
  *** Deduced "t2" physically_standard_tape yes using rule p1.

  Is it a 7- or 9-track tape
  (Give one of: 7, 9,  or unknown).?
x.
  *** Valid responses are members of either;
      the set of variable values:  {7, 9},
      or the set of keywords:  {u, w, s, t, nt, an, nan, x}.

  Is it a 7- or 9-track tape
  (Give one of: 7, 9,  or unknown).?
nan.
  *** The Goal Annunciator is now off.

  Is it a 7- or 9-track tape
  (Give one of: 7, 9,  or unknown).?
9.
```

```
  What is the recording density (in cpi)
  (Give one of: 800, 1600, 6250,  or unknown).?
1600.

  What is the record length (in bytes)
  (Give an integer between 1 and 32768 or unknown).?
80.

  Are the records blocked
  (Give one of: yes, no,  or unknown).?
yes.

  Are the records of fixed or variable length
  (Give one of: variable, fixed,  or unknown).?
fixed.

  What is the block size (in bytes)
  (Give an integer between 18 and 32768 or unknown).?
1600.

  What is the character code
  (Give one of: ebcdic, ascii, bcdic,  or unknown).?
ascii.

  What is the estimated CPU time (s)
  (Give an integer between 1 and 600 or unknown).?
unknown.

  I have written the following JCL programme to read your tape:

//ABCC  JOB ,,
//      NOTIFY=ABC,
//      TYPRUN=HOLD,
//      MSGCLASS=T,
//      CLASS=1
//*   ********************************************************************
//*   *** JCL GENERATED BY TAPE-READING EXPERT SYSTEM      ***
//*   *** ON 09/18/84 AT 14:03:42                          ***
//*   ***    R. F. DANCER,                                 ***
//*   ***    E.R.L.,                                       ***
//*   ***    SEPT 1984.                                    ***
//*   ********************************************************************
//COPY   EXEC PGM=IEBGENER
//SYSUT1 DD UNIT=TAPE1600,VOL=SER=ABCT20,DISP=OLD,
//       DCB=(RECFM=FB,LRECL=80,BLKSIZE=1600,
//            DEN=3,OPTCD=Q),
//       LABEL=(,NL)
//SYSUT2 DD UNIT=SYSDA,DISP=(NEW,CATLG),
//       DSN=ABC.LOOKSEE.DATA,
//       SPACE=(1600,(50,150),RLSE)
//SYSPRINT DD SYSOUT=*
//SYSIN    DD DUMMY
//
```

```
 This code has been copied to file abct20aa.

 *** Concluded "t2" result task_complete using rule c1.
 *** Terminating examination for "t2".
yes
 >
dt.
 *** Execution time was 35129 mS.
yes
 >
call(stats).

Prolog Execution Statistics:
Local Frames: 14, Variables: 10
Global Nodes: 11
Heap Nodes: 23954
Trail Entries: 2
Atoms: 897, Chars: 10257

yes
 >
end.

 *** Leaving ACSI.
{Leaving Prolog}
READY
```

DOCUMENT CONTROL DATA SHEET

Security classification of this page: UNCLASSIFIED

1 DOCUMENT NUMBERS

AR Number: AR-004-140

Series Number: ERL-0330-TR

Other Numbers:

2 SECURITY CLASSIFICATION

a. Complete Document: Unclassified

b. Title in Isolation: Unclassified

c. Summary in Isolation: Unclassified

3 TITLE

DIESS - A PROLOG IMPLEMENTATION OF A DOMAIN INDEPENDENT EXPERT SYSTEM SHELL

4 PERSONAL AUTHOR(S):

R.F. Dancer

5 DOCUMENT DATE:

May 1985

6 6.1 TOTAL NUMBER OF PAGES 29

6.2 NUMBER OF REFERENCES: 18

7 7.1 CORPORATE AUTHOR(S):

Electronics Research Laboratory

7.2 DOCUMENT SERIES AND NUMBER

Electronics Research Laboratory 0330-TR

8 REFERENCE NUMBERS

a. Task: DST 83/213

b. Sponsoring Agency:

9 COST CODE:

10 IMPRINT (Publishing organisation)

Defence Research Centre Salisbury

11 COMPUTER PROGRAM(S) (Title(s) and language(s))

12 RELEASE LIMITATIONS (of the document):

Approved for Public Release

Security classification of this page: UNCLASSIFIED

Security classification of this page: UNCLASSIFIED

13 ANNOUNCEMENT LIMITATIONS (of the information on these pages):

No limitation

14 DESCRIPTORS:

a. EJC Thesaurus Terms: Computer programming

b. Non-Thesaurus Terms: DIESS Version 1

15 COSATI CODES:

09020

16 SUMMARY OR ABSTRACT:
(if this is security classified, the announcement of this report will be similarly classified)

A software environment suitable for the expression of expert knowledge in a variety of domains has been designed and implemented in the Prolog computer language. This 'shell' is intended to be capable of containing expert systems whose object is the proof of goal symbols in domains of categorical single-valued facts.

An example is given of a non-trivial expert system to programme in the IBM Job Control Language. Some conclusions are reached and recommendations given for future developments.

Security classification of this page: UNCLASSIFIED